Rigorous Component-Based Development

Alan Cameron Wills, Trireme, alan@trireme.com

Desmond D’Souza, ICON Computing

Keywords: Re-use, Catalysis, framework, contract-based, collaboration, specification, refinement, rigorous method, component-based development

Popular OO methods provide intuitively attractive diagramming notations, but mostly lack clear semantics and do not support a well-defined way of composing models. In this paper we outline the key constructs from the Catalysis method, and show how they build upon each other to provide a basis for framework-based modeling and design, based upon:

- type models: a basis for defining a type
- collaboration: a partial definition of how typed objects interact when playing roles
- refinement: of both individual types and of collaborations
- frameworks: a generic unit of modeling or design
- framework collaborations: a particular kind of framework that utilizes placeholder types and generalized actions to permit flexible composition

Authors’ address: Trireme International Ltd,
24 Windsor Road,
Manchester M19 2EB
UK

Phone: +44 161 225 3240

Fax: +44 161 257 3292

Email: alan@trireme.com

DRAFT: This paper is behind schedule!
If accepted for publication, it will be improved to remove some inconsistencies and make it more succinct.

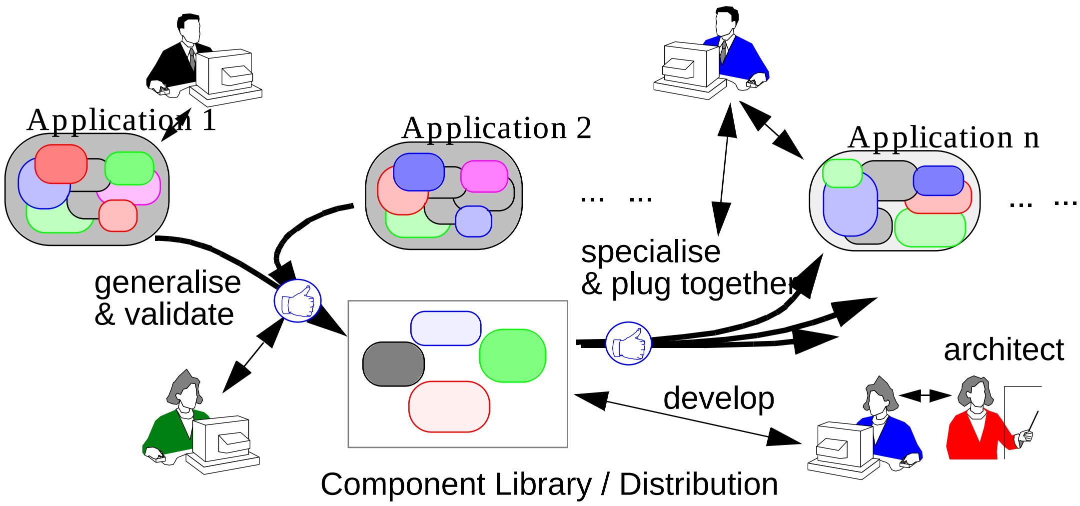
Word Count: 8994

1 *Building systems with components*

One of the big motivations for taking up OO design is that it promises some form of re-use. The naïve interpretation of “re-use” is that you can pull pieces out of an old program and install them in a new one; but that doesn’t often work very well. A more mature strategy is to look for similarities in different pieces of design work (perhaps in the same system, or in the same project, or in the same organisation) and to expend some resources on creating (and subsequently refining) a generic piece of design, or ***framework***, that covers all the similar cases; then it’s likely that subsequent designs will be able to instantiate the generic framework. Much experience has shown that this results in faster more maintainable designs, if less efficient end-products.

More generally, ***components*** may be any coherent unit of design effort that can be packaged, sold, kept in a library, assigned to one person or team to develop and maintain, and re-used. Components can be classes or frameworks; or objects that can be dynamically plugged at run-time; high-level designs; specifications; patterns; extensions to existing components; or even project plans.

Any design or model can be seen as a component composed of smaller ones. We believe that in the future, there will be considerably more emphasis on building and distributing components, and building systems from them.



1.1 Components need tools need formality

Component based design needs tool support. But tools cannot provide much useful help with text and diagrams that have no precise meaning: so informal language is not good enough. Since our definition of a useful component includes more than just program code — indeed, some of the most widely useful are patterns — we therefore need a sufficiently formal language in which to express composable specifications, high-level designs, patterns, and so on — as well as programs.

Even for components that do take the form of program code, it is necessary to define interfaces — that is, to say what one component expects of whatever other it is connected to. Lists of operations (as in IDL, Java interfaces, etc) are not really sufficient for this purpose, because they don't include information about what the operations are expected to achieve. In a closed programming environment, it may be feasible to plug components together by conferring with your colleagues informally; but this is not feasible when components may be distributed worldwide: then, more effort must be spent on unambiguous specification.

1.2 Catalysis: component-based design

Catalysis [2] is an approach to design compliant with the UML notation, which incorporates several features we believe are important for component-based development. It has been developed over the past few years by the present authors, working with a number of our clients, with whom it has proved successful. One of our larger industrial collaborators, itself with a large customer base in development support tools, is currently basing its next generation of tools on this method.

In an industrial context, it is important that any development approach is practical. It must be exactly as rigorous as it needs to be, and no more so. In the past, rigorous methods — precise specification of requirements and documented refinements — have not proved widely popular. But a component-based strategy shifts the economics of development towards more investment in each component, if each will be re-used.[1]

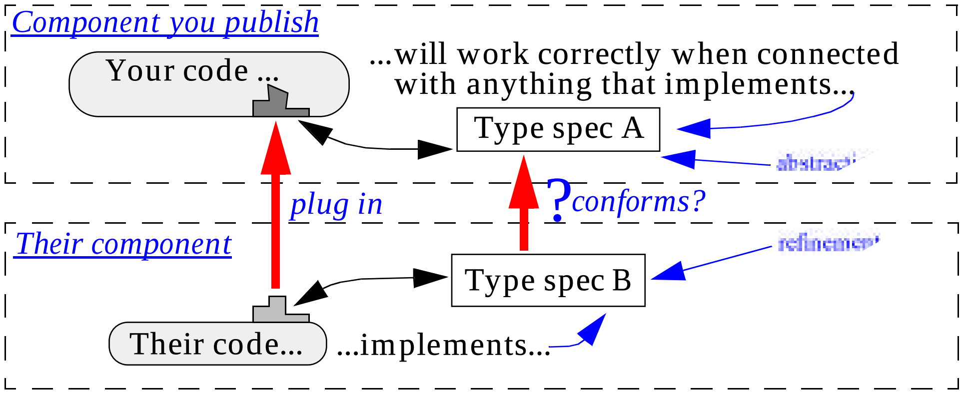
Above all, we emphasise abstraction and conformance. Abstraction means you describe just the properties you require from a development, or at an interface. Conformance means that a particular more detailed description — whether more detailed analysis or an implementation — is truly described by the abstraction; and further, it means you document why you believe this to be so.

Abstraction means you can layer a development, separating the more important decisions from the lesser; and lets you describe many variations in a basic decision at one go. The process of refinement means making decisions and showing how they conform to the earlier ones.

Writing abstractions precisely (in a well-defined notation rather than the more traditional ad hoc/natural language style) has two benefits: it avoids misunderstandings between designers; and the act of writing them unearths inconsistencies and gaps that traditionally remain undiscovered until coding. Abstractions and conformance can be documented in various degrees of precision, from the loose sketch to the highly formal argument: but the principles — their underlying semantic structure and metamodel — are the same.

1. We also adopt various techniques to make the formal style more approachable. For example, statecharts and other diagrams are used where possible in preference to plain predicates and postconditions. See [2].

In component-based design, clear abstraction is essential as the means to specify the contract between a component and any others that may be used in conjunction with it. Conformance is the essential relationship between an abstract



Big Question. When some third party puts these components together, will they work? What should you do to be sure they will work?

requirement and any realization. Documenting conformance means expressing the belief that a more specific requirement (anything down to a complete implementation) is correctly described by a more general abstraction: it should come with some justification for the belief — whether a sketched illustration or a complete signed-off test suite.

1.3 Key concepts of Catalysis

The remainder of this paper describes Catalysis in terms of four key features. Each is presented with an example, and a summary of the benefits gained. These form a basis from which a great variety of patterns of modeling and refinement can be built [2]. The first two will be familiar to those used to precision in abstract modeling techniques; collaborations and frameworks are perhaps more novel, and add an important degree of expressive power.

Briefly, the key features are:

- Types — abstracting behavior using type models.

  A type defines a set of objects by their externally visible behavior, without describing an implementation. Precise description of behavior needs an abstract model of the state of any correct implementation of that type — a type-model. A type is not the same as a class.

- Conformance and refinement — a foundation for traceability.

  A conformance is a relationship between two descriptions (types, collaborations, type and class, etc.) where one claims to conform to the guarantees of the other. It is accompanied by a mapping that justifies this claim. There are several kinds of conformance, the simplest being between a type and a class that implements it. A significant part of a greenfield development process consists of *refining* a design — that is, creating a series of extensions

and transformations that ultimately can show the implementing code to conform to the highest-level requirements abstraction (though not necessarily produced in top-down order!) Re-engineering consists in first abstracting an existing design to a more general requirement, and then refining it to a new design e.g. with new features or better performance.

- Collaborations — abstracting joint behaviors and dialogue.

  The most interesting aspects of design and architecture involve partial descriptions of groups of objects and their interactions relative to each other. A collaboration defines a set of actions between typed objects playing certain roles with respect to other objects in that collaboration. A collaboration provides an abstraction of multi-party interactions and of detailed dialogs between participants. It provides a unit of scoping i.e. constraints and rules that apply within vs. outside the group of collaborators; and of conformance i.e. more detailed realizations of joint behavior.

- Frameworks — a foundation for design composition and re-use.

  Specifications, models, and designs, all show recurring patterns of abstract structures and behaviors. The key to such patterns are the relationships between elements, as opposed to individual types or classes. An application of such a pattern specializes all the elements in parallel and mutually compatible ways, as opposed to an individual specialization of each element. A framework defines patterns in generic terms by utilizing *placeholders* for elements. It can be applied to a family of related types to *generate* different models and designs. A framework may be a model or group of models for a collaboration; or a static relationship between objects; or a single type; or a class; or a package of related classes.

Underlying all of these is a clear semantics: every diagram can be translated into an equivalent text form; and every statement in our notation is defined in terms of a well-defined model of objects and the interactions between them [2].

2 *Types and Behavior Specification*

2.1 Definitions and Example

Type. A type defines a set of objects by their externally (client) visible behaviors. Any object which conforms to those behaviors is a member of that type and is usable as such by that client, regardless of the class(es) used to implement it. A type formalizes the signature-based notion of an *interface*.

Types/interfaces are defined for the benefit of clients or collaborators. The type which describes the behaviors required by a client must match (either exactly, or via some conformance abstraction) the type which describes the services provided by the implementor.

Behavior Specification. In order to define a type we need a precise description of its behaviors. This description must permit any correct implementation of those behaviors, hence it must be in implementation-independent terms. However, it must still be precise enough to prescribe the required correctness criteria. One practical way to document behaviors is to specify each operation in terms of pre/post conditions (anywhere between informally and precisely, and including state diagram notations). The contracts these represent may be just documentation, or may form the basis for a practical test suite.

To describe the behavior of any object we have to say what happens to its state. This almost always requires an underlying *vocabulary* which describes the abstract state of any implementation object.

In many cases the state is directly visible externally, through a read-only subset of the of the operations. For example, good vending machines tell you how much money you've already put in, and the postcondition of inserting a coin is to increase that visible attribute. Business objects often adhere to this rule of direct visibility of all attributes. But this is not always the case. A bad vending machine (we've all met them!) doesn't display the amount inserted, and you yourself have to count carefully. But the attribute is still there in a very real sense: clients have to be aware of it (or can infer it from observed behavior) in order to understand the machine's response to the operations they perform.

So documenting attributes is necessary for accurate specification of the object — they give you a vocabulary in which to express your pre and postconditions.

This vocabulary can be made precise by introducing a set of *hypothesized* typed queries on that type, and then describing the actions of that type in terms of their effect on those abstract attributes (e.g. using pre and post-conditions). Queries correspond to attributes and associations in OOA.

Type Model. A model of a type (i.e. an abstraction of any implementation of that type) as a set of abstract queries or attributes, in terms of which its behaviors can be specified. Type models range from simple to complex, and may be depicted textually or graphically. Basing our notation on UML, we depict the type model in the *middle* section of a UML box — the place traditionally reserved for attributes.

The example (a) below illustrates a very simple type, **Counter**. We can *model* the effect of *inc*, *dec* in terms of a *count* attribute. How should we *type* this attribute? It depends completely on what behavioral statements we want to make that depend on this attribute. Since we want to increment and decrement this attribute, it is *convenient* to describe it in terms of *int*, and use what we already know about ints. This does not mean that there must be a *stored integer instance variable* called count. One fairly absurd implementation would use a *List* object which supported a *length* function, adding a random object to the list for every *inc*, and removing one for every *dec*.

The next example (b) illustrates a **CashRegister** type[1] supporting *startSale, addItem, deleteLastItem,...*. When we write down the behavior specifications of these operations we realize that we must discuss many new terms:

- a set of available products, *products* — so we can say what happens when you enter a known vs. an unknown product;
- the price of every known product, *priceOfProduct(Product)* — so we can accumulate totals;
- the quantity of each item within a sale, *quantity(Item)* — so we can describe the cost of that line item;
- the sequence in which the items were rung up within this sale — so we can define the effect of *deleteLastItem*), etc.

These may be formalized as a set of queries as shown below. Of course, there is no implication that these represent stored data, or even any internal private methods. Rather, we only require that any correct implementation should exhibit the listed operations, and that a client's expectations of their behavior, as understood from this model, must not be disappointed. This in turn means that there must be a clear mapping to the model's queries from the variables or fields of the chosen implementation.

1. Either the real kind, or one in cyberspace, used by some software agent or a human

```
<<type>> Counter
---
count: integer
---
inc
  count == old(count) + 1
dec....
```

(1a)

```
<<type>> CashRegister
---
products: Set<Product>
priceOfProduct (Product): Money
itemsOfProduct (Product): Set<Item>
quantity(Item): integer
cost(Item): Money
itemsOfSale (Sale): Seq<Item>.......
currentSale: Sale
allSales: Seq<Sale>
inv currentSale    allSales
---
startSale () -- a new Sale has been made the currentSale
addItem (Product, quantity)
deleteLastItem () -- the last item of the currentSale has been deleted
closeSale () -- currentSale has become the last Sale in allSales; paid
listSaleHistory ()
```

(1b)

Figure 1

```
CashRegister::closeSale        // a sample operation specification
    post currentSale == old(allSales.last)
```

2.1.1 Types membership and models

A type is a set of object-histories: member ship is determined by an object's externally visible behaviour, and whether it is a member of a particular type is fixed throughout its life (though it may also be useful to talk about sets of which it may be a member temporarily, dependent on changing properties). An object is in general a member of an infinity of types, since there are always more sufficiently loose specifications you can write.

We write x:T to signify that object x conforms to the behaviour represented descriptions of type T. A type is usually specified by providing an abstract model of its state — that is, a set of attributes — over which operations are specified using pre and postconditions. (We also use framing clauses and rely/guarantee conditions, none of which are illustrated in this paper. Immutable objects such as numbers are specified in an algebraic style. The integration of model-based and algebraic styles is similar to Larch [8].)

The occurrence of an attribute in a model (such as currentSale:Sale) means that for any object cr which is a member of the CashRegister type, the expression cr.currentSale is defined and refers to an object which is a member of the Sale type:

cr : CashRegister · (cr.currentSale) : Sale

The attributes form an abstract model. There is no implication that objects in the type have any features of these names either internally or visibly. The only requirement is that the operations listed in the bottom section of the box should exhibit the behaviour implied by the model. It is sometimes helpful to think of the attributes as functions that *could* be implemented for any implementation, perhaps for test purposes.

Attributes with parameters are not always found in modelling languages, but we have found them a powerful tool.

2.1.2 Pictorial presentation

We can depict any of the queries visually as UML associations: notice the correspondence to Figure 1b. Each association defines a pair of model queries. The choice between attribute and pictorial presentation of a query is a stylistic one.

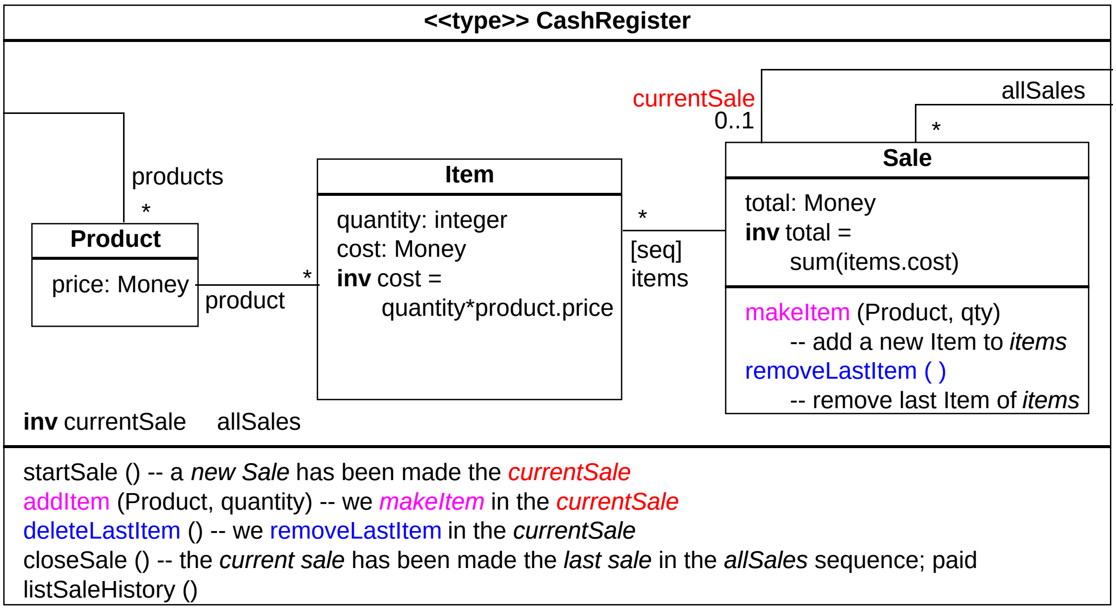
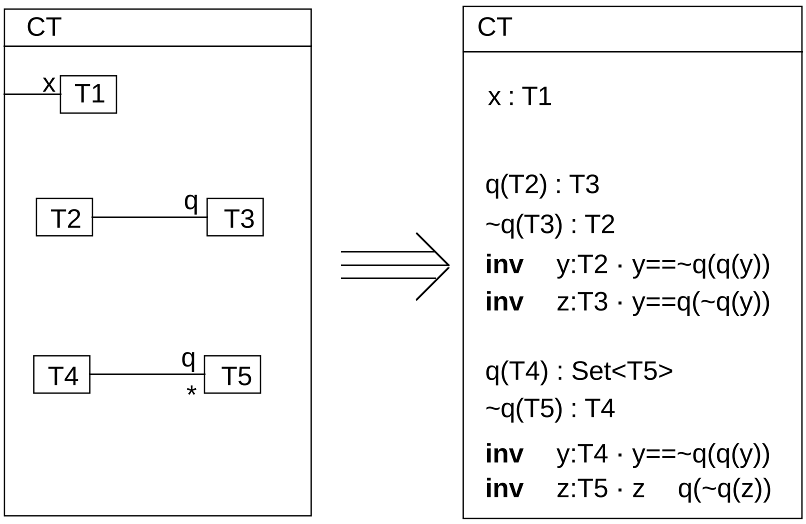


Figure 2

Each assication in the diagram can be translated to the earlier style of attributes using a number of straightforward rules:



The inclusion of operation-specs in model types does not pre-empt design: since the types shown inside the box are strictly only a model, they might not appear in every implementation. However, they are a convenient way of breaking up or *factoring* the specification of the component into smaller, type-dependent pieces.

2.1.3 Where does the model come from?

It is usual to create the model of a system or component pictorially, basing it on a semantic model of the 'business', the concepts and relationships understood by the system's users. Operations performed by or upon the objects in that world become reflected in the system model.

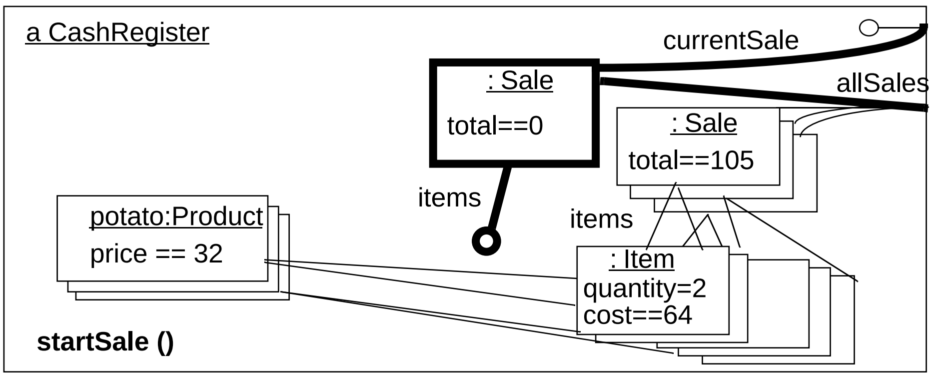
In some design styles, the business model immediately becomes the basis for the design; modifications are then made to improve performance, re-engineer-ability, etc. We separate these two stages: we build a specification model based on the business concepts; and then, separately, build a design to meet that spec.

There are two motivations for this separation. Firstly, if you are specifying a component's interface, based on a model of the concepts about which the interlocutors will communicate, you just don't know what the implementation will look like: there may be many. Secondly, even if you are specifying something that you are then going on to design yourself, we have found that it is very valuable to keep a separate specification as a formalised version of the normal requirements document. The activity of formalising what would otherwise be expressed in natural language and ad-hoc pictures, tends to expose gaps and inconsistencies very effectively.

Notice that strictly, we are only specifying the CashRegister here, not any internal parts. We defer less important decisions — in this case about internal structure. It is of course a tenet of OO design that the internal structure should in fact follow the business model as far as possible — but we allow that an interface which has a very abstract spec may be met successfully by many designs; and that what makes a readable specification may be an inefficient design, so that localised refinements must be made.

2.2 Snapshots

Along with the more pictorial presentation of a model goes a very useful illustrative and pedagogical device for envisioning the effects of operations from their pre/postcondition specs. A *snapshot* is a picture of part the state of an example system. The current values of associations are drawn as links. Two or more can be drawn on top of each other to show the effect of a particular operation. Here, links and objects new after the action startSale are drawn bold:



2.3 Problems Addressed

Decisions deferred. Type models and behavior specifications abstract away implementation structure and algorithms without loss of precision.

Behavior-Driven vs. Data-Driven approach. Our approach to defining a type solves the long-standing controversy between behavior-driven and data-driven methodologies. UML, in particular, has taken some criticism from the behaviorists for what is perceived as a data-centric approach. Using our approach this becomes a non-issue. A highly simplified description is:

- Start with the behavior of a component. Let's look particularly at CashRegister's *deleteLastItem* operation for an example.
- To define behavior precisely you will need an underlying vocabulary — which should relate to clients' concerns rather than any implementation. Formalize this model in terms of a set of queries.
  In this case, we say that there is such a thing as a current Sale; and that every sale has a sequence of items: this allows us to talk of deleting the last item of the current Sale.
- Draw the queries as a type-model diagram where needed. i.e. using type boxes with attributes, associations, and invariants.
- Factor parts of the behavior specifications into the types within the main type model. *removeLastItem* in *Sale* helps define *deleteLastItem*. This allows specifications to be easily restructured and re-used. Nevertheless, there is still no mandate that the outer type, the CashRegister, must be implemented internally using classes with these operations: everything inside the outer box is just a model, and implementation considerations may deviate the designer from this structure.
- Do not show any queries unless they are needed to help express some behaviors of the object of interest.
- Any correct implementation will have some mapping from its concrete representation choices to the queries in the type-model. The type model is *not* a concrete data model. Rather, it simply defines an abstract (yet precise) set of terms for expressing behaviors.

(See http://www.iconcomp.com/papers/data-vs-behavior/index.html for a more detailed discussion of this issue.)

Observations. In component-based design, it is essential to document what is required of other components you may connect to, and whose implementation you know nothing about. You can think of a type specification as a test-suite for checking conformance of any implementation to your requirements.

A type specification should suffice to separate implementations that fulfill the requirements from those that do not, while imposing no more requirements than necessary. This is the key difference between the notions of 'type' and 'class': a class embodies an implementation that will have various incidental features, not guaranteed to be supported in later upgrades; a type says exactly what is guaranteed, and is silent about what is not.

A complete application can be characterized by a type; or a small software object; or a real-world object; or one role of an object; or one of many interfaces to an object.

When you design and implement an object it exhibits more types than you intend, since its different (unanticipated) clients will use different aspects and combination of features. A class implements many types — as many as you like, and more than you've thought of[1]. Suppose you design and implement a **lamp** type, fulfilling the requirement of providing light in exchange for electric power. Someone else may use it simply to drain their notebook battery — your implementation also conforms to the **powerdrain** type that represents their requirement! But if your type guaranteed only light, you could rightly implement an oil lamp; the laptop owner would be well advised to look elsewhere.

A Paradox? Some see a paradox here that is worth clearing up. By the principle of encapsulation, clients should not depend on the inner structure of a design; the client should be able to work with any provider that exhibits the required behavior externally. Attributes are about the state of an object: does it not violate encapsulation to use them in type specifications?

No. Attributes *model* (abstract) state — they describe features that can be conceptually ascribed to it, but they do not need to represent actual internals. Every queue has a length, which increases on *append* (among other effects). In an array implementation, there may be a variable that directly reflects the length (or perhaps length+1); but in a linked-list implementation, it may have no direct realization: you retrieve it by counting the nodes. Any satisfactory implementation should give no surprises to a client who understands the spec: there can be all sorts of complex stuff going on underneath the hood.

1. We are abstracting differences in language / compiler checking of types. Some languages require that all permitted type "views" of your class are defined before the class itself.

3 *Type Conformance*

3.1 Definitions and Example

Clients do not know, and should not depend on, how a query from a type-model is implemented. Every implementation will do this differently. The information must be there, but it doesn't matter how or under what name.

Type Conformance. One type (the *realization* or *implementation*) conforms to another (the *abstraction*) iff the behavior clients gets from the realization always fulfills the expectations set by the abstraction. A class conforms to a type iff the behavior it prescribes will always fulfill the type description.

Refinement. Refinement means creating or choosing a conformant type or class, and documenting why you believe it is conformant. You might write a spec and then implement it; or you might find a component in a library; or a library component may require a plug-in to be supplied by you.

Documenting a refinement (in any of these cases) means writing down:

- The reasons you believe the supposed abstraction really does describe the implementation accurately.
  This is an invaluable sanity check and helps reviewers and maintainers.
- The reasons for choosing this implementation from alternatives (so those that follow won't fall down the same pits you fell into along the way).

Retrieval. A mapping which "*retrieves*" the abstract model from an implementation, helping justify the conformance claim. For example, the implementation may be highly optimized and perform all manner of complex caching; the abstract model might simply refer to some abstract state. We must be able to "*retrieve*" the abstract state terms from any correct implementation.

For example, I might define a **lamp** type to use electrical energy and produce light. You might define a **powerDrain** type to deplete a battery. To claim that **lamp** is a refinement of **powerDrain** the retrieval must map the usage of electrical energy to the depletion of a battery (two different vocabularies).

A key part of the refinement is to document how each attribute is realized — e.g. write each model attribute as a function within the implementation[1]. It doesn't matter how slowly they execute, as it may well be unnecessary to use them in the code. The point is that the exercise of writing them documents and demonstrates how the state information represented by abstract attributes is realized. From a practical point of view, they can be used as part of a test strat-

1. Traditionally written as pure functions, retrievals may also be written as executable functions, to assist program testing. Informal retrievals are also permitted, even to a simple *"abstractQuery = ask John to explain"*!

egy, since the postconditions will be written in terms of the abstract attributes. (For example, Eiffel includes a mechanism for the execution of pre and post-conditions.)

A type may be *defined* as an extension of another type. In this case, the behavior specifications and type model of the super-type are inherited by it — there is no need for any explicit retrieval. The subtype may extend the behavior by defining new operations, or by providing *additional* specifications of inherited operations — the specifications simply conjoin with the inherited specs. i.e. there is no *overriding* of behavior specifications (in contrast to the traditional overriding of method implementations for a subclass).

Class. A description of the internal design of an object: in program code, the instance-variables and function bodies. In programming, an object is created as an instance of a particular class. Some languages (like 'self') do without classes; but their designers still need types.

In a programming language, we may think of a class as implementing some consciously designed types (which interact with client objects), and additionally implementing a low-level type which represents its interface to the virtual-machine which is the target of the language compiler.

Since the type(s) that a class implements had a behavior specification, the class needs to meet that specification. The type utilized a *type-model* to provide the vocabulary to define the actions. The class chooses certain *instance variables* for its implementation. A *retrieval* is a mapping from the concrete to the abstract. It establishes traceability and the correctness of the implementation e.g. a design review would check the retrievals.

The figure below shows a type called SelectionList, which might be used in the implementation of user selections for our CashRegister. It allows items to be added, deleted, selected, and de-selected. The type model for specifying these operations includes a concept of count, which item is at which position, and which items are selected. There is also a convenience effect called *movedUp*, which could be defined completely in terms of the other queries.

We also have a class named ListBoxA, and the retrieval to the type. As you can see, the retrieval is a property of the *implements* relation between class and type. In general, it is a property of any *refines* relationship, establishing the mapping between two type-models (the queries and their corresponding types) and the corresponding sets of actions.

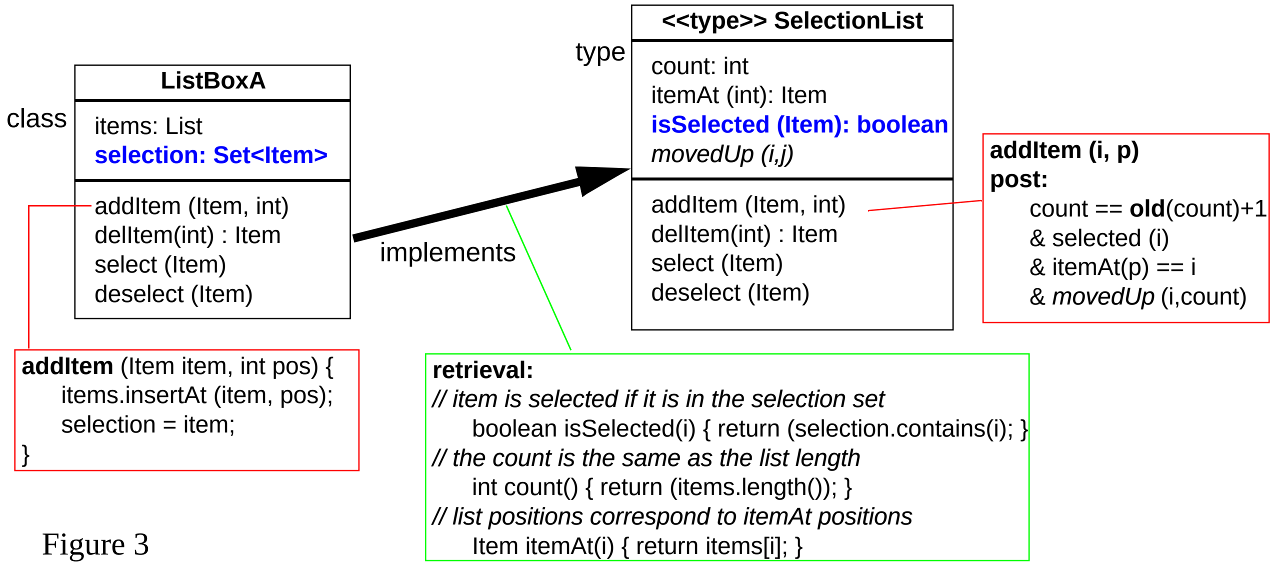


Figure 3

A similar retrieval could be defined for an implementation of *CashRegister* in Figure 2. For example, we might implement it using a sequence of sales (the type model uses a set *allSales*, and a distinguished *currentSale*). The head of that sequence corresponds to (*retrieves* to) the abstract query *currentSale.*

The retrieval is only part of a full conformance check, which should properly include a proof that each of the abstract operation specs is implied by those of the implementation. In practice, once the retrieval is written down, conformance can often be clearly demonstrated informally; or can be checked to a satisfactory extent with a test harness 'executing' the pre/postconditions.

3.2 Problems Addressed

Refinements and retrievals give a precise basis for traceability (aiding understanding, reverse-engineering, and system maintenance and extension), and substitutability (aiding team-work and parallel development). In addition, they solve a very real problem in software development today, particularly in light of the move towards iterative development and incremental delivery i.e.

> *"I have just made some change to my code. Do I have to update my design models? My analysis models?"*

Our rule is very simple:

> Given multiple levels of abstraction, propagate the change to the highest level that is invalidated by your change. This means that the abstract levels must be defined precisely enough to be refuted, even if informally.

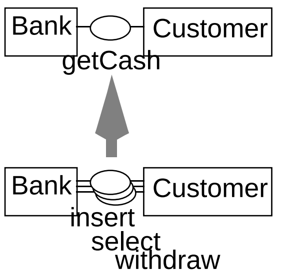
Subclass. A class whose definition is partly derived from another class, its superclass. Methods may be overridden. The superclass-subclass interface can be defined using the techniques of types and type-models.

4 *Collaboration, Actions, and Refinements*

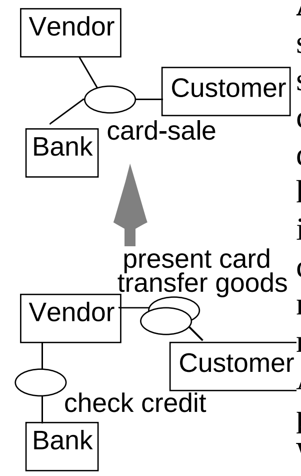
Types are the syntactical unit of specification; collaborations are our unit of object-oriented design. A collaboration indicates what objects participate in an interaction, and allow the outcome to be specified. In line with our goal of abstraction, collaborations permit layering of decisions about how the responsibilities are distributed between the participants; and about the detailed protocol of the interaction.

4.1 Definition and Examples

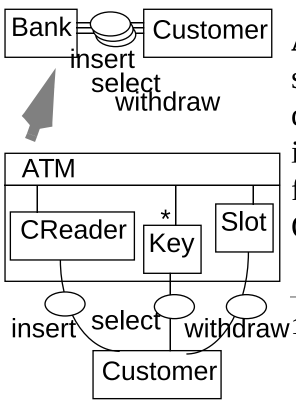
The most interesting aspects of design and architecture involve partial descriptions of groups of objects and their interactions relative to each other.



A collaboration abstracts detailed dialogue or protocol. In real life, every action we talk about — for example “I got some money from the cash machine” — actually represents some sequence of finer-grained actions e.g. “I put my card in the machine; I selected ‘cash’; I took my money and my card”. Any action can be made finer. But at any level, there is a definite postcondition. A collaboration spec expresses the postcondition at the appropriate level of detail. (“There’s more cash in my pocket, but my account shows less.”). Thus we defer details of interaction protocols.



A collaboration abstracts multiple participants. Pinning an operation on a single object is convenient in programming terms, particularly for distributed systems; but in real life — and at higher levels of design — it is important to consider all the participants in an operation, since its outcome may affect and depend on them all. So we abstract operations to “*actions*”. An action may have several participants, one of which may possibly be distinguished as the initiator[1]. For example, a *card-sale* is an action involving a buyer, seller, and card-issuer. Likewise, we generalize action-occurrences, as depicted in scenario diagrams, to permit multi-party actions, as opposed to the strictly sender-receiver style depicted by using arrows in sequence or message-trace diagrams. A standard OOP operation (= message) is a particular kind of action. The pre/post spec of an action may reflect the change of state of all of its participants. We can thus defer the partitioning of responsibility when needed.



A collaboration abstracts object compositions. An object that is treated as a single entity at one level of abstraction may actually be composed of many. In doing the refinement, all particpants need to know which constituent of their interlocutor they must deal with. For example, in abstract “I got some cash from the bank” — actually, you got it from one of the bank’s cash machines. Or in more detail, you inserted your card in the card-reader of the cash

1. Function-calls, rendezvous, hardware signals, messages are all varieties of action. So are transactions, use-cases, sessions, remote procedure calls, joint-actions, and complete dialogues. We will later explain how this clarifies the (over)usage of use-case in UML 0.91

machine, pressed its "cash" key, and took the money from the exit slot. The machine and the bank turn out to be abstractions: there is no bank at all, just a conglomeration of interrelated machines and people!

Collaboration. A collaboration[1] defines a set of actions between typed objects playing certain roles with respect to other objects in that collaboration. The actions are specified in terms of a type-model shared by all collaborators. The actions may be *joint* (responsibility not assigned to a role) or *localized* (responsibility assigned), and may be *internal* or *external* to the collaboration. Each role is a place for an object, and is named relative to the other roles in the overall collaboration. As a degenerate example, in a single action collaboration, role names are parameter names, and participant types are parameter types.

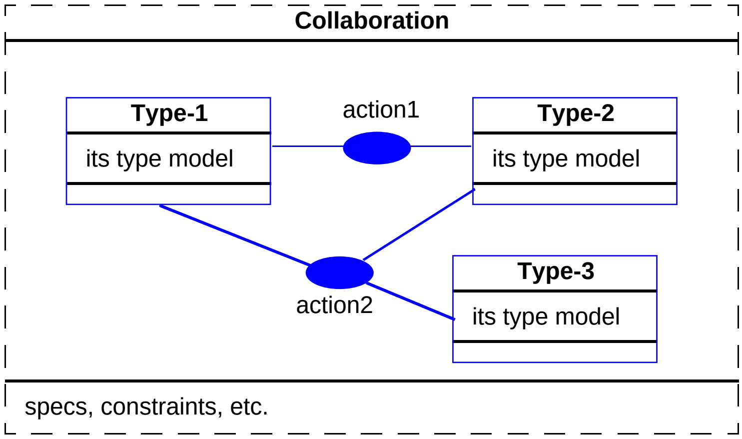


Figure 4

Collaboration refinement. A collaboration may be refined by a more detailed set of actions and objects, either joint or localized.

4.2 Retail Example

Many useful forms of refinement do not result in sub-types, but rather represent a refinement of the *collaboration* itself i.e. all involved parties are affected. The example below illustrates this for multiple levels of abstraction in a buyer-seller interaction, from an abstract action *purchase(Set<Item>)*, to a more specific dialog of first *pickup(Set<Item>)* followed by *pay(Sale)*, to the

1. A collaboration spec makes precise the notion of a *role-model* used in OORAM [5]

detailed level of *startSale(), addItem(),... endSale()* (which in practice could be done with the CashRegister as part of the model). This refinement affects all participants, and a user-manual for the cash register documents the retrieval!

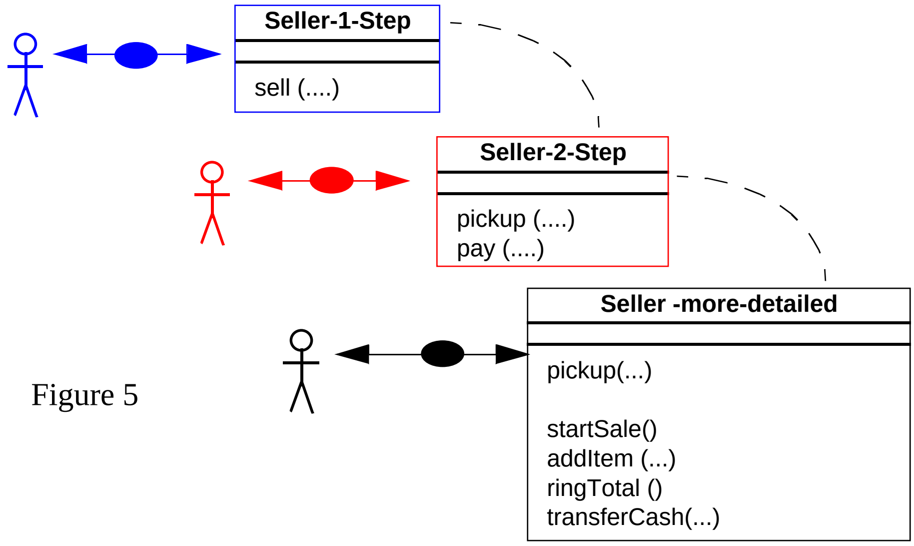


Figure 5

Although this illustration is cast in user interface terms, the same technique can be applied to layer descriptions of the interfaces between software components. For example, the buyer might be a Java software agent.

What we are doing here is refining the entire collaboration — not the individual types — with an accompanying mapping (including a *retrieval* of type models) to justify the conformance claim. Note that *Seller-2-Step* is not a subtype of *Seller-1-Step.* A person (or software agent) expecting Seller-1-Step would not work successfully with its counterpart Seller-2-Step, which supports a different set of actions.

Once again, a design review would focus attention on the refinement. A collaboration refinement usually entails two different aspects:

- Firstly, the finer actions must be related to the more abstract ones: what sequences of finer actions induce each abstract one? State charts are good for specifying the sets of action sequences that correspond to each abstract one, and interaction diagrams (sequence or graph form) to illustrate specific scenarios. The pre/post specs of permitted sequences of finer operations should combine to fulfill those of the abstract actions.
- Secondly, at a finer level of detail, more detailed type-models of each participant are needed. For example, when some sale items are picked up first, the seller must know the amount of the sale due (and possibly the selections) on the subsequent payment. Hence the refined type model needs

extra attributes for this, in addition to the more abstract level that just dealt with stocks and money.

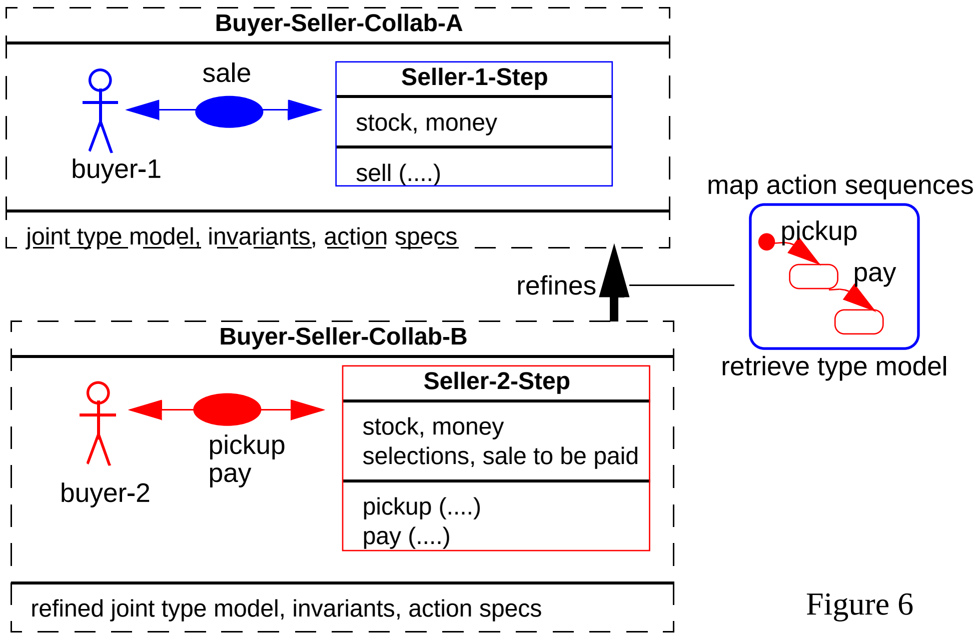


Figure 6

The extra states and the state charts can conveniently be reified in an object whose purpose is to represent the abstract action[1]. Real-life constructs such as purchase orders represent reifications of ongoing refined actions. Much leverage can be obtained by reifying a collaboration-refinement as an object in a spec, whether or not it is implemented directly in a design. For example, to deal with concurrency, to handle rollbacks, invariants between the participants, and so on. This usually corresponds to the OOSE usage of a *control object.* Often this can become a real object in a design — the supervising object for that use-case. If this is not suitable to implement, the designer has the option of treating it as an abstract model that can be implemented differently.

There is often an argument about the boundary between analysis and design, or specification and implementation. Does a complete specification describe every detail of the object's interface, down to the API level, even to the parameter-passing conventions? In theory, yes; in practice, we might simply accept that further "design" detail affects both the client as well as provider i.e. the collaboration.

In combination, collaborations, actions, and refinement provide a solid basis for defining use-cases and traceability. Hence, it is important that a *Collaboration* be a first-class model element subject to its own scoping and refinement rules, and not just a diagram!

1. This is compatible with UML 0.91. It adds a clear semantic reason based on refinement, and provides a basis for the levels of abstraction in a layered development process that all methods claim few support at a deep level.

4.3 Problems Addressed

- Abstracting interactions and transactions.
- Describing and packaging partial join object behaviours.
- Bridging the gap from the use-case to the message-send.

Why not simply utilize use-cases? A *use-case* in UML 0.91 covers too many concepts. The original use-case in Jacobson's OOSE is a very useful construct. Fundamentally a complex *action*, it packages together the following: a single abstract *action* (a simple collaboration) which achieves some business objective for a participant, a more detailed collaboration which realizes the effect of that action, a refinement relation between the latter and the former, and the set of scenarios which help understand the refined collaboration. A use-case offers significant value by virtue of its focus on business tasks, and we should retain that. A collaboration is *not* just an action. It would be a mistake to fail to separate out the more fundamental constituent parts of a use-case: action, collaboration, and refinement.

5 *Frameworks and Composition*

A Catalysis ‘framework’ is a formally-described generic package of development work. Frameworks can be composed to make specification models, high-level designs, as well as program code. Composition of designs is illustrated here — more detail can be found in [8] and [2].

5.1 Definition and Example

Each collaboration partially defines the joint behaviors of some sets of objects. Due to its partial nature, it is essential to be able to compose these collaborations. This is the basis of defining and applying *frameworks* to a design. The

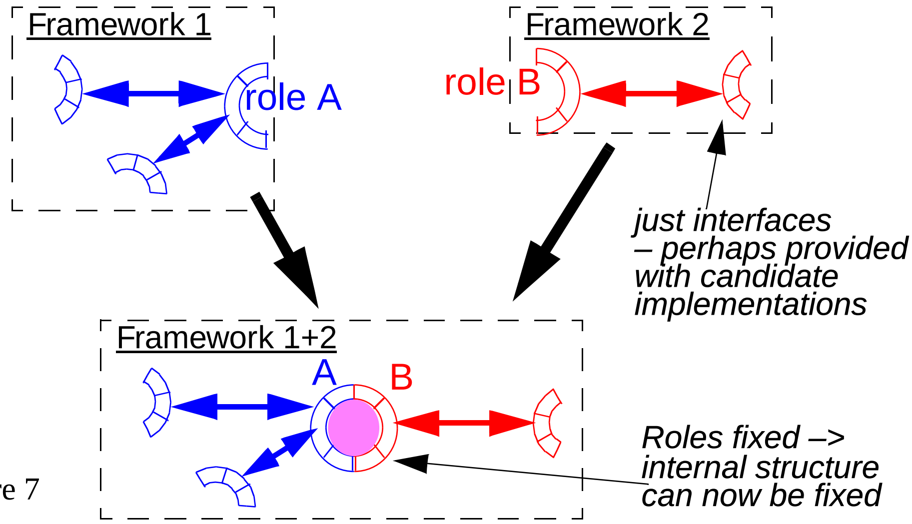
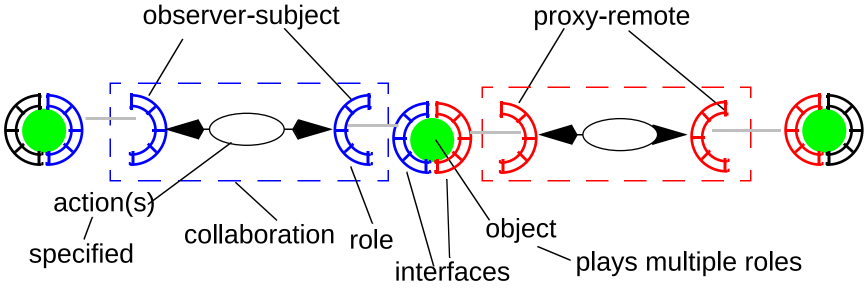


Figure 7

figure below illustrates a composition of the well-known subject-observer and proxy-remote design patterns. For example, a *sale* object might play the role of a *subject* with respect to an *sale window*, and simultaneously play the role of a *proxy* with respect to an *sale data* object. Frameworks are usually generic and parameterized collaborations that can be easily composed.

Composing systems from frameworks



This goes significantly beyond the parameterisation of individual classes as in C++. To reiterate, the most useful pieces of design are about the (static or dynamic) relationships between things, rather than about single objects. The crucial decisions in OOD are about the distribution of responsibilities between parties to collaborations; and how they work together to achieve some overall goal. Looking at a Book of Patterns, most of what we see is about several rather than one object i.e. about collaborations. We want to effectively capture, re-use, and compose these pieces of design.

A framework is a generic description of a piece of development work, often a model or group of models for a collaboration; or a static relationship between several objects; or a single type; or a class; or a package of related classes. The programming-language concept it most nearly corresponds to is the package; but it may be a chunk of implementation, or a piece of abstract specification. In maths terms, it is a theory. In a framework-based metamodel, all development work is done within the context of some framework.

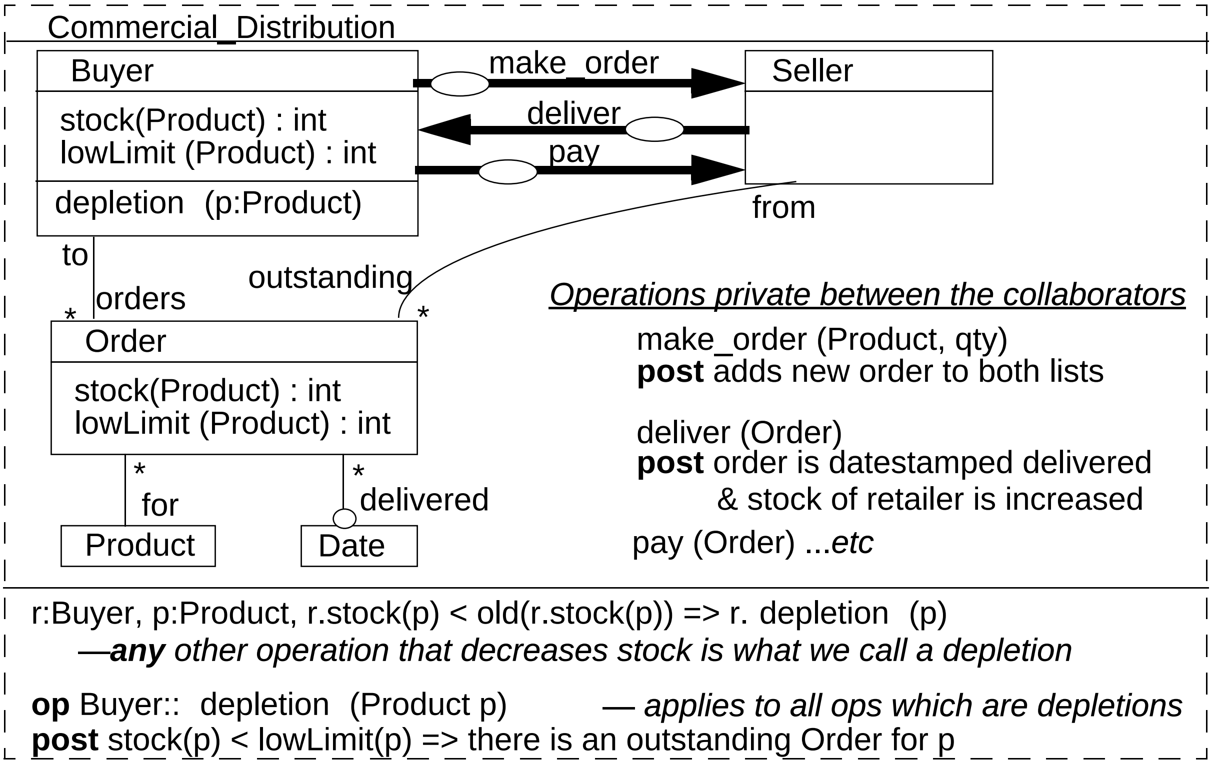
Since it focuses on the relationships between objects as opposed to a particular object, a framework will in general contain just a partial view of each object — the role it plays in that relationship. By talking about, for example, *business frameworks* rather than *business objects*, we can deal with a more general and interesting class of modeling and design problems.

To build a complete design (or specification,...) we compose frameworks. In doing so, we often bring together several views of an object, and the composed framework will combine these roles. Each framework may impose both requirements and restrictions that appear in the result.

For example, we may have a framework, *Commercial_Distribution,* about how commercial companies buy things from one another on wholesale, with the system of purchase orders, distribution, delivery notes, invoices, and the corresponding interactions. Another framework, *Retail_Sales*, may be about how casual retail customers buy things, with the options of credit cards, cash, and so on. Some types of object — shops — may play roles from both these worlds, with distributor-based re-ordering and retail selling. The actions from both frameworks affect the shop's state.

Framework abstraction. Encapsulating and making generic a piece of modeling or design in one logical unit. These are rarely single types or classes; more generally, they are about the roles played by several objects.

To describe a partial view of one object, it is sometimes necessary to talk about the effects of unknown actions from the object's other roles. So for example, it is part of the retail sales collaboration that we order more stock when we are running low. In the commercial distribution framework, we describe the collaboration that gets a purchase order issued and eventually fulfilled, but do not specify what triggers the decrease in stock. We don't know, since the framework might be applied to many different objects that maintain stock levels and use it for different purposes, such as to recover from spoilage, or from thievery.



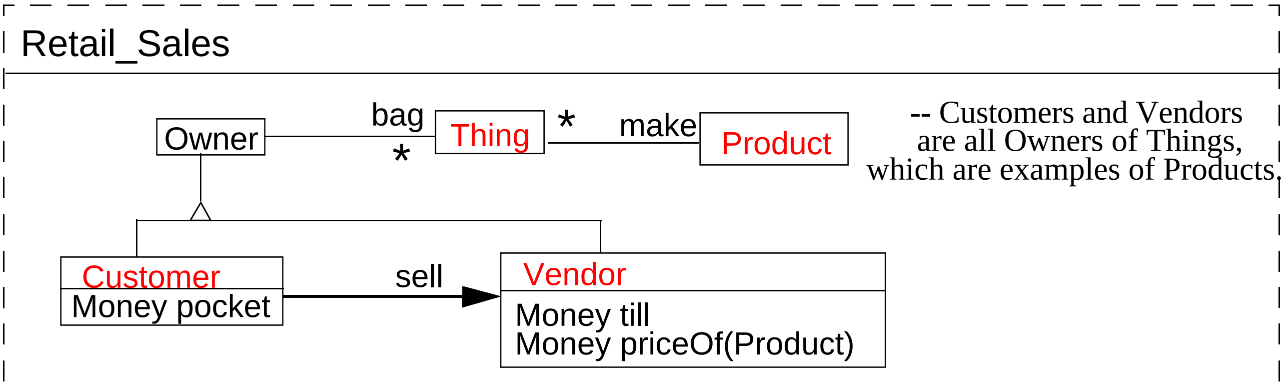
Framework with abstract operations

So we describe the unknown action by its effects, and impose the constraint that whatever causes those effects — in some eventual implementation — must also trigger our restocking procedure. In the example, the placeholder action depletion represents any action that causes a reduction in stock.

Such hooks are a characteristic of frameworks. They may describe restrictions on behavior otherwise permitted by the other roles which will be composed in the same object. For example, a precondition may be strengthened to disallow [whatever operation causes] stock depletion, whenever stock is beyond a certain age. Action-specifications from different frameworks are ‘joined’, a covariant composition whereby both preconditions and postconditions are separately conjoined [2].

Framework refinement. A framework can be specialized in a variety of ways — by parameterization; by specialization (a form of inheritance); or by composition with other frameworks. Framework composition works by applying several frameworks to one set of types (Figure 7), so that their definitions come partly or wholly from the partial definitions in the frameworks.

A framework is usually defined with *a set of placeholder types*. The example below illustrates this for a simple retail-sales collaboration that has been made into a generic framework. The types in **<...>** highlight placeholder types.



op c:Customer –> v:Vendor::sell(Product p) -- sell is sent by Customers to Vendors

post t:Thing, t.make==p
& c.pocket –= v.priceOf(p) & c.bag += t
& v.till += v.priceOf(p) & v.bag –= t

-- When a Customer asks a Vendor to sell a Product, the Customer's pocket is depleted by the Vendor's price for it; the Customer gets a Thing of that make;
Meanwhile, the vendor gets the money but loses that Thing.

Figure 9

To apply a framework, placeholder types with their attributes and actions are substituted by real types or placeholders of the importing framework.

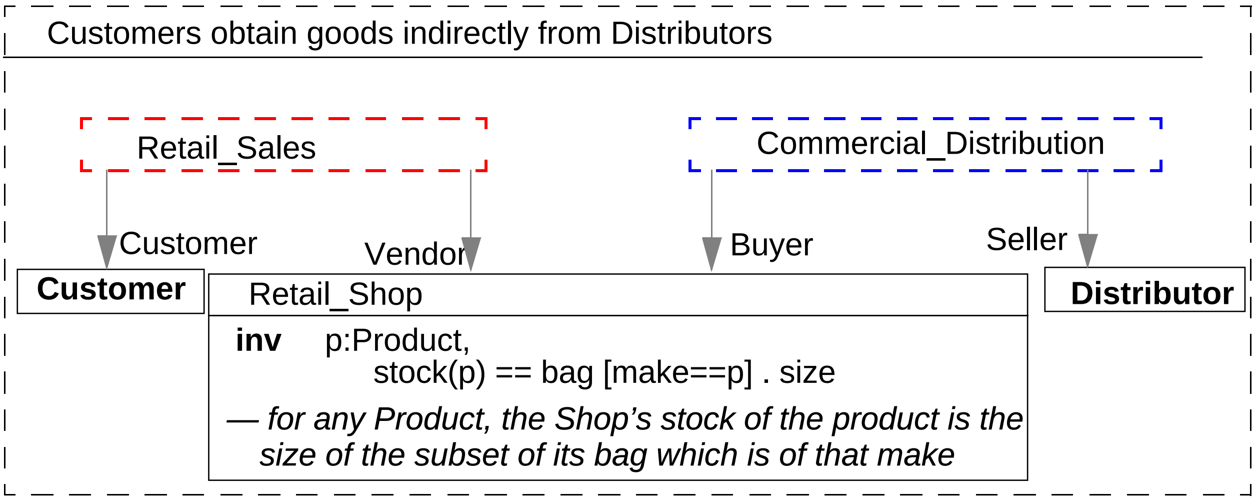


Figure 10

The above example illustrates how two frameworks, *Retail_Sales* and *Commercial_Distribution*, both defined with some placeholder types, can be *applied* to define a new framework in which a *Retail_Shop* is substituted for the *Vendor* and *Buyer* placeholders, while still leaving placeholder types for *Customer* and *Distributor*. The two source frameworks had some type-models with attributes for their placeholder types. The composition relates these together using an *invariant*.

Decisions deferred. Other roles of an object; how one role affects another.

Observations. A parameterized framework is the formal core of a pattern.

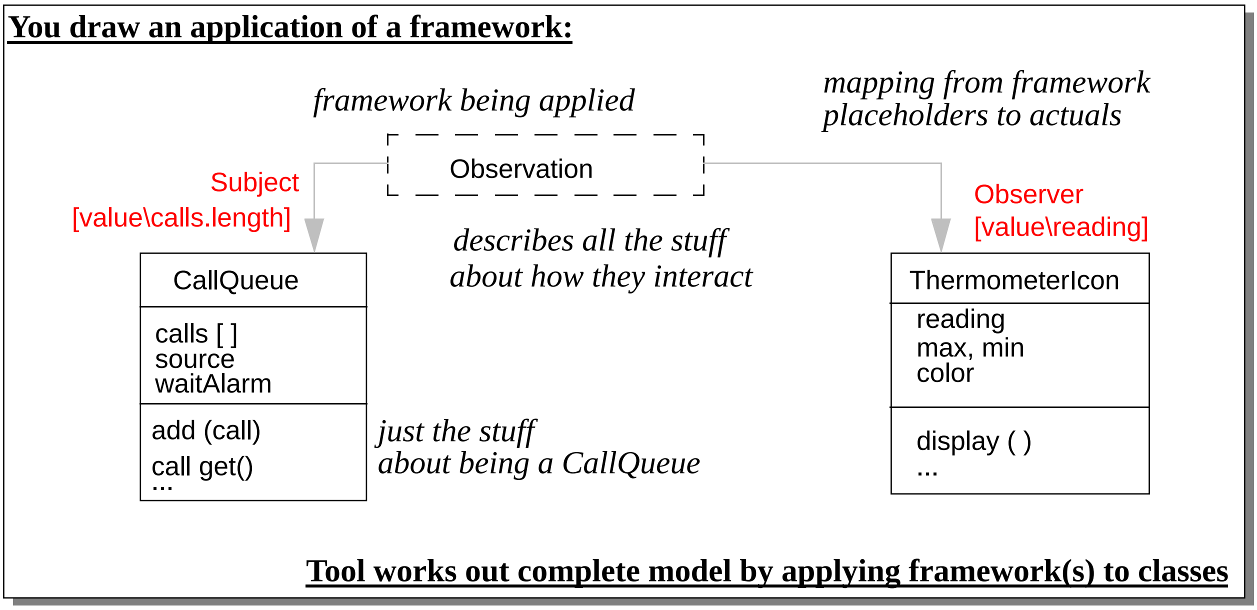


Figure 11

This figure illustrates another example of applying the *Observation* framework (frequently described as *Subject-Observer*), showing some of the substitutions for attributes in the framework type-model — *subject.value* of a placeholder *Subject* corresponds to *calls.length* of the *CallQueue*. Thus, the thermometer icon *reading* is constrained to track the *length* of the *CallQueue*, following the behavior imposed by the abstract framework. Our precise modeling not only reduces ambiguity, but enables tools that do more than just draw diagrams — a good tool could work out the complete resultant model from the application of a framework.

In each case, the types in the refined result are not, in general, subtypes of their counterparts in the abstract framework. This is because the actions and invariants from one framework usually impose *constraints* on the objects they are applied to. In most OO frameworks, collaboration refinement — which affects all participants, and does not produce individually substitutable subtypes — is the rule rather than the exception. The example below illustrates the point for a *subject-observer* collaboration. You cannot use a *SaleWindow* observer with a *PowerSwitch* subject, hence it is *not* a subtype of *Observer.*

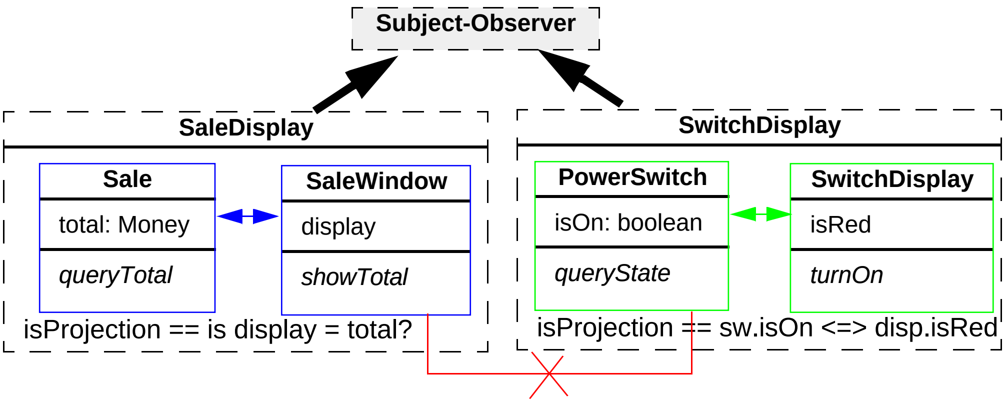


Figure 12

5.2 Problems Addressed

This section achieves the following:

- Composing partial views of collaborative behaviors.
- Takes a significant step towards supporting frameworks/patterns.
- Takes a significant step out of a subclass and inheritance-centred view towards frameworks.

Why not make use-cases generic? By trying to be everything to everyone, the use-case risks losing its relatively clear original intent. A framework is *not* just an action; a use-case is fundamentally an action (together with a refining collaboration, scenarios, etc.). Frameworks are a distinct and powerful generic construct to adopt into a modeling language.

6 *Summary*

The key features of Catalysis are directed at:

- Component-based development —where ‘component’ doesn’t just mean ‘object’. Frameworks may represent interactions between object roles. Interfaces may be precisely defined with types.
- Layered development. Types abstract behaviour of individual objects from design; collaborations abstract overall goals of an interaction from detailed protocol. Layers are traceably connected with refinements.

6.1 Ongoing work

Notable gaps at present are support for real time; and support for concurrency. However, there is good work going on in these fields, and we hope to integrate some of it.

In collaboration with industrial and academic partners, tools are being developed to support types, collaborations and their refinements, frameworks and their composition.

7 References

[1] D. D'Souza, *Behavior-driven vs. Data-driven Methodologies: A non-Issue?* http://www.iconcomp.com/papers

[2] D. D'Souza and A. Wills, *Component-Based Development with Catalysis*: To be published mid-1997.

[3] UML 0.9 Documentation Set, http://www.rational.com

[4] G.Leavens et al, *Object specification using Larch*

[5] Reenskaug, Wold, Lehne: Working With Objects, Manning/Prentice Hall. New York 1996

[6] D. D'Souza and A. Wills, *Extending Fusion: Practical Rigor and Refinement*: http://www.iconcomp.com/papers; also in *Fusion in the Real World*, D. Coleman et al eds, Prentice Hall, 1995

[7] B Meyer, *Object oriented Program Construction,* PHI 1988

[8] A. Wills, *"Frameworks",* in proceedings of *Object-Oriented Information Systems,* London Dec 1996. Also http://www.trireme.com/papers/fworks.html

[9] S.Cook & J.Daniels, *Designing Object Systems*, [PHI 1994]. Discusses abstract models, refinement and strong semantics for OOA models.

[10] D.Coleman et al, *Fusion,* 1994. Discusses abstract models and refinement.